

510.84

IL6r

no.1139-1145

1983-84

INC.

cop.3

510.84
IL6r
NO.145

Report No. UIUCDCS-R-83-1145

UILU-ENG 83 1723

The Delay/Re-Read Protocol for
Concurrency Control in Databases

by

M. Dennis Mickunas
Pankaj Jalote

March 1983

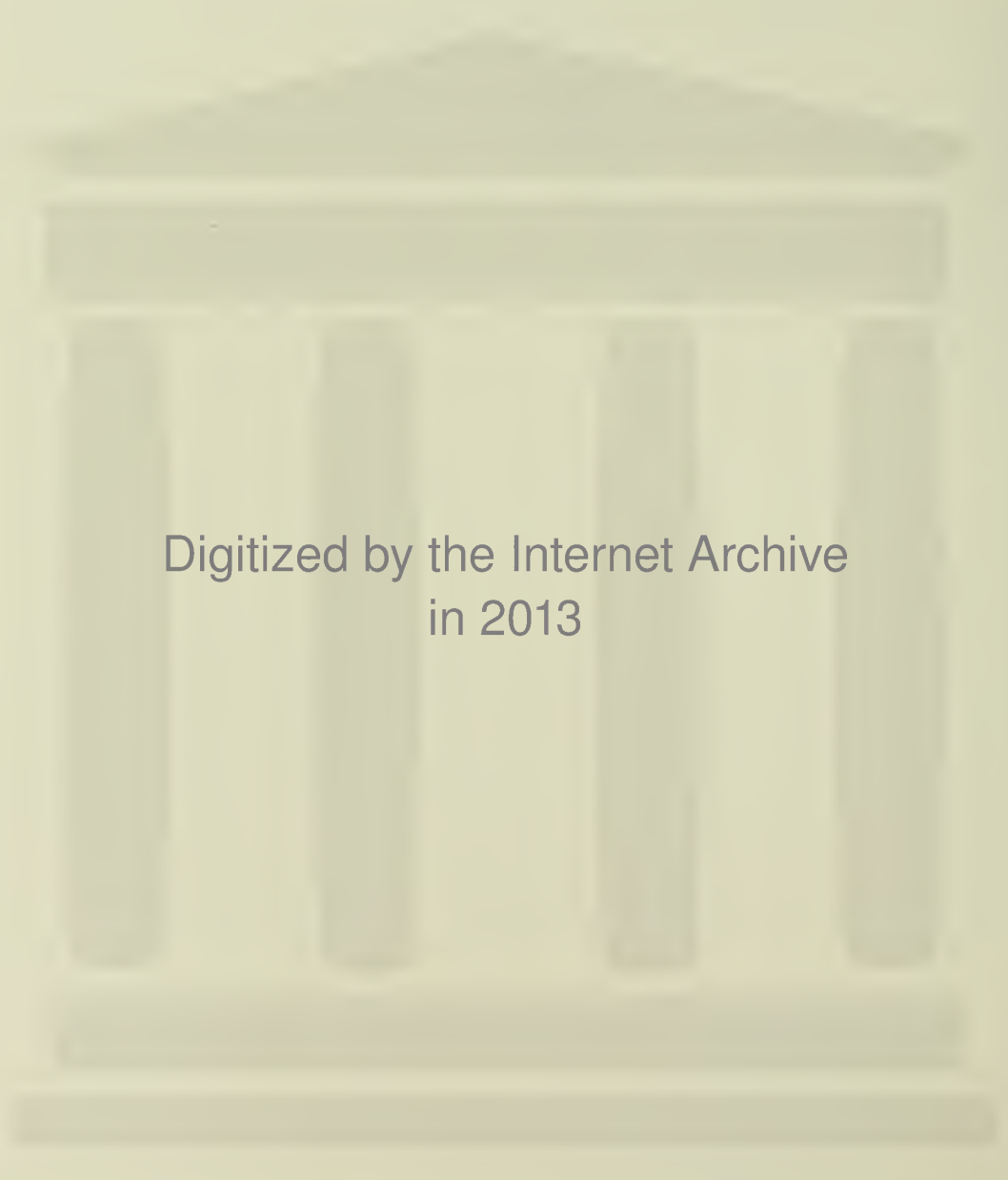


DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE LIBRARY OF THE

FEB 22 1984

UNIVERSITY OF ILLINOIS
URBANA-CHAMPAIGN



Digitized by the Internet Archive
in 2013

<http://archive.org/details/delayrereadproto1145mick>

The Delay/Re-Read Protocol for Concurrency Control in Databases

M. Dennis Mickunas
Pankaj Jalote

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

March 28, 1983

*Research supported in part by Highly Available Database Project,
IBM Research Labs.

ABSTRACT

We present a new protocol, called the Delay/Re-Read Protocol, for controlling concurrent access to a database. The protocol uses a combination of preventative and corrective measures for maintaining consistency. The Delay/Re-Read Protocol is deadlock-free, requires no backup data, and does not abort either transactions or database writes.

1. INTRODUCTION

The problem of concurrency in databases has received a good deal of attention in recent years. Errors resulting from unrestricted concurrency were first observed by Eswaran et.al. [ESW]. They showed that unrestricted concurrency can result in an inconsistent database. The solution proposed was that each transaction *lock* the entity it is going to access. Moreover, such a locking protocol typically adopts so-called "two phase locking", which consists of a "growing phase", and a "shrinking phase". In the growing phase a transaction collects the locks that it requires, and in shrinking phase it releases them. A transaction cannot request any further locks once it has released any lock. A disadvantage of two-phase locking is that it is not deadlock-free.

Since then many variations on locking have been proposed [BAYb, ELL], and it has been demonstrated that locking achieves somewhat better results when the database is structured as a hierarchy [KED, SIL]. However, the use of locking to maintain consistency is an entirely *preventive measure* i.e. it tries to prevent any view of the database from becoming inconsistent. For this reason locking assumes the worst case and is often overly restrictive. In an effort to relieve the tight restrictions of locking protocols, Kung [KUN] proposed a *corrective measure* for concurrency control in which a transaction is permitted to see an inconsistent state, but its view will be later 'corrected' and made consistent.

In the present paper we present a new protocol, which utilizes both preventative and corrective techniques. The protocol, which we call the *Delay/Re-Read Protocol*, acts, on the one hand, in a *corrective* fashion by sometimes forcing a transaction to re-read some data; it does so upon recognizing that a transaction has read an inconsistent set of data. The protocol acts, on the other hand, in a *preventative* fashion by sometimes imposing a delay before permitting a transaction to write to the database; it does so upon recognizing that such a write might, at the present time, jeopardize the integrity of the database.

This paper is organized as follows. In Section 2 we present our model of a database system. In Section 3 we define our notion of consistency and present some results relating consistency to the ordering of basic actions of transactions. In Section 4 we present the Delay/Re-Read Protocol and prove that it is both consistent and deadlock-free. In Section 5 we discuss some aspects of the Delay/Re-Read Protocol, including its application to distributed databases.

2. SYSTEM MODEL

We consider the database to be a collection of distinct named objects, called *entities*, together with assertions about the values of the entities. These assertions, called *integrity constraints* are often not explicitly states; indeed, the space required for full specification of all integrity constraints might exceed that required for the database itself. Nonetheless, whether explicitly stated or not, consistency constraints are present and they govern the interactions of *operations* upon entities. A database which satisfies all of the integrity constraints is said to be in a *consistent* state.

In order to formalize our model, we present some definitions.

We denote the set of entities in the database by "E". Such an entity may be read or written *indivisibly*.

Definition 2.1. A *transaction* with unique transaction number k , denoted T^k , is a set of *actions*

$$T^k = \{t_i^k\}_{i=1}^{p_k}$$

together with a linear ordering¹ $\leq_{T^k}$, on T^k . As a notational convenience, we use the subscripts on the actions to reflect this linear ordering, viz. $i < j$ implies $t_i^k <_{T^k} t_j^k$. For each $i \in [1, p_k]$, t_i^k is a 4-tuple

$$t_i^k = (k, a_i^k, e_i^k, U_i^k)$$

where

- (1) k uniquely identifies the transaction to which t_i^k belongs
- (2) $a_i^k \in \{R, W\}$, called the *operation*, denotes either Read or Write
- (3) $e_i^k \in E$ denotes the entity upon which the operation a_i^k is performed
- (4) $U_i^k \subseteq 2^E$, called the *use set*.

□

Clearly, $<_{T^k}$ is meant to reflect the temporal ordering of the individual actions of T^k . The *use set* U_i^k of $t_i^k = (k, a_i^k, e_i^k, U_i^k)$ is the empty set if $a_i^k = R$; consequently, we may often omit U_i^k when writing a Read transaction:

$$t_i^k = (k, R, e_i^k)$$

In case $a_i^k = W$, then U_i^k denotes the subset of entities which are used to compute the new value of e_i^k ; consequently, we may often use a "function" notation when writing a Write transaction:

¹ Recall: a *partial ordering* $\leq$ on a set X is a subset $\leq \subseteq X \times X$ for which $(a, b) \in \leq$ and $(b, a) \in \leq$ implies $a = b$, and for which $(a, b) \in \leq$ and $(b, c) \in \leq$ implies $(a, c) \in \leq$; $(a, b) \in \leq$ is usually written $a \leq b$; if $a \leq b$ and $a \neq b$ then we write $a < b$; $\leq$ is said to be a *linear ordering* on X if for every $a, b \in X$, either $a \leq b$ or $b \leq a$.

$$t_i^k = (k, W, e_i^k(U_i^k))$$

Our model imposes some additional restrictions upon a transaction, *viz.*

- (1) all Reads must precede all Writes;
- (2) the *use set* for a Write transaction must include the entity being written, i.e. the new value of an entity is a function of its old value (among other things), and;
- (3) an entity must be read before it can appear in the use set of any Write.

Formally, we have the following.

Definition 2.2. A transaction $T^k = \{t_i^k\}_{i=1}^{p_k}$ is said to be *well-formed* if and only if the following conditions hold:

- (1) for each pair of actions $t_i^k, t_j^k \in T^k$, $a_i^k = R$ implies either $i < j$ or $a_j^k = R$
- (2) if $t_i^k = (k, W, e_i^k(U_i^k))$ then e_i^k is in U_i^k
- (3) if $t_i^k = (k, W, e_i^k(U_i^k))$ then for every $y \in U_i^k$ there exists t_j^k ($j < i$) for which $t_j^k = (k, R, y)$.

□

Our model is thus a generalization of Papadimitriou's "two-step restricted" model [PAPb], in which our restrictions (2) and (3) with $U_i^k = \{e_i^k\}$ reduce to the simpler restriction that an entity must be read before it can be written.

3. CONSISTENCY

We assume that a given transaction, T^k , transforms the database from one consistent state to another consistent state (although the database may temporarily be in an inconsistent state while T^k is ongoing). Our goal is to allow multiple transactions to be active concurrently, and yet to ensure that the database will finally end up in a consistent state.

The notion of concurrent execution of multiple transactions is captured in the following.

Definition 3.1. Let $T^1, \dots, T^n$ be transactions. A *schedule*, S , for $T^1, \dots, T^n$ is the set of actions

$$S = \bigcup_{i=1}^n T^i$$

together with a linear ordering, $\leq_S$, on S , for which $\bigcup_{i=1}^n \leq_{T^i} \subseteq \leq_S$.

□

As before, the relation $<_S$ is meant to reflect temporal ordering (with truly simultaneous actions having an "effective" temporal ordering imposed by $<_S$). Since we are interested in only those schedules which correspond to the (possibly concurrent) execution of the individual transactions T_n^k , it follows that if $t_i^k <_{T^i} t_j^k$ then we must have

$t_i^k <_S t_j^k$; hence the requirement that $\bigcup_{i=1}^n <_{T^i} \subseteq <_S$.

The aim of any concurrency control method is to allow only those schedules which transform the database from one consistent state to another. Serializability [ESW, PAPA] has been generally accepted as the consistency criteria for schedules. Informally, serializability holds that a schedule for transactions $T^1, \dots, T^n$ is consistent if the state of the database after executing the schedule is the same as it would have been had the transactions been executed one after another in some order. Note that the order (corresponding to some permutation $\{\pi_i\}_{i=1}^n$ of $[1, n]$) is not specified.

Given a schedule, S , which satisfies this notion of consistency, we refer to the permuted serial execution $T^{\pi_1}, \dots, T^{\pi_n}$ as an *Equivalent Serial Schedule (ESS)*. Such an ESS is not necessarily unique.

Since our model requires that each entity be read before it can be written, a schedule S can be checked for serializability using its corresponding *precedence graph*, G_S [ULL]. We construct the graph as follows. The nodes correspond to the transactions. The arcs are determined by the following rule:

If $(i, R, x) <_S (j, W, x)$ or $(i, W, x) <_S (j, W, x)$ or $(i, W, x) <_S (j, R, x)$ for any x , then draw an arc from T^i to T^j .

A schedule S is serializable if its precedence graph is acyclic. It follows that we can find ESS(s) for S by *topological sorting*.

We call the first write by a transaction T , the *critical write* (denoted W_c) of T . Note that $(i, R, a) <_S (i, W_c, b)$ for any T^i .

Lemma 3.1. Let $T^1, \dots, T^n$ be well-formed transactions and S a schedule for which $1 \leq i < j \leq n$ implies $(i, W_c, a) <_S (j, W_c, b)$. The precedence graph G_S has no arc from T^j to T^i ($i < j$) if for every (i, W, x) in S , $(j, R, z) \in S$ implies either $z \neq x$ or $(i, W, x) <_S (j, R, z)$.

Proof. There are only 3 possible ways that G_S can have an arc from T^j to T^i :

(1) $(j, R, x) <_S (i, W, x)$. By hypothesis, $(j, R, x) \in T^j$ and $j > i$ implies $(i, W, x) <_S (j, R, x)$ which, by anti-symmetry of $<_S$, disallows this case.

(2) $(j, W, x) <_S (i, W, x)$. Since T^j is well-formed, we have

$$(j, R, x) <_S (j, W, x)$$

whence as in case 1

$$(i, W, x) <_S (j, R, x)$$

and

$$(i, W, x) <_S (j, W, x)$$

which, by anti-symmetry of $<_S$, disallows this case.

(3) $(j, W, x) <_S (i, R, x)$. Since T^i is well-formed,
 $(i, R, x) <_S (i, W_c, a)$

By definition of critical read,

$$(j, W_c, b) \leq_S (j, W, x)$$

whence

$$(j, W_c, b) <_S (i, W_c, a)$$

However, by hypothesis (since $i < j$)

$$(i, W_c, a) <_S (j, W_c, b)$$

so this, the final case, is also disallowed by anti-symmetry of $<_S$.

Therefore there can be no arc in G_S from T^j to T^i .

□

Theorem 3.2. Let $T^1, \dots, T^n$ be well-formed transactions and S a schedule for which $1 \leq i < j \leq n$ implies $(i, W_c, a) <_S (j, W_c, b)$. The precedence graph G_S is *acyclic* if for every (i, W, x) in S and for every $j > i$, $(j, R, z) \in S$ implies either $z \neq x$ or $(i, W, x) <_S (j, R, z)$.

Proof. The proof is by contradiction. Suppose that G_S has a cycle involving nodes $T^{i_1}, \dots, T^{i_k}$ ($k > 1$). Let $i = \min\{i_1, \dots, i_k\}$. Now, for every $j \in \{i_1, \dots, i_k\}$ ($j \neq i$) we have $j > i$, so Lemma 3.1 applies, and there can be no arc in G_S from T^j to T^i . Therefore, the presumed cycle involving T^i is contradicted.

□

Corollary 3.3. Let $T^1, \dots, T^n$ and S be as in Theorem 3.2. Then S is consistent.

Proof. The proof is immediate since G_S is acyclic.

□

Informally, the theorem lays down the condition to be satisfied by the schedule such that every transaction sees a consistent state, i.e. the set of values returned by the Reads of the transaction is such that it is the same as the set of values of these entities in some consistent database state. This does not imply that all the Reads must be performed on the same consistent state. A Read can be performed on any database state, possibly transitory and inconsistent, but the set of values read by all Reads must be such that all the values *can co-exist* in some consistent database state. Theorem 3.2 specifies the condition when this is satisfied. This theorem is the basis of Delay/Re-Read Protocol.

In the following sections $W_i(x)$ and $R_i(x)$ mean same as (i, W, x) and (i, R, x) respectively.

4. DELAY/RE-READ PROTOCOL

Not all schedules satisfy the condition of Theorem 3.2. The purpose of the Delay/Re-Read Protocol is to take *any schedule* and "correct" it such that the schedule satisfies the condition of the theorem. Each transaction is submitted to a *Transaction Manager* which assigns a *Transaction Process (TP)* to each transaction. Each TP submits its Read and Write requests to a process called the *Concurrency Control Process (CCP)*; the TP awaits permission from the CCP to proceed with the requested Read or Write. There is also a *History File* which only the CCP reads or writes. This is different from a "log file" which might record all activity as well as "backup" data. The History File records only a finite window of activity. The CCP, on receiving a Read request, merely records it in the History File (since we exert no control over the Reads). However, a Write can be delayed by the CCP according to the following scheme.

The Delay/Re-Read Protocol is used by the CCP to ensure that any schedule remains consistent. This is accomplished by a combination of preventative and corrective measures. On one hand, the Delay/Re-Read Protocol sometimes causes the CCP to *delay* the Critical Write of a TP, thereby delaying all of the Writes of the associated transaction (a preventative action); on the other hand, the Delay/Re-Read Protocol sometimes causes the CCP to instruct a TP to *re-read* some entities prior to proceeding with a Write, thereby assuring that the Use Set for the Write is consistent (a corrective action).

Since we do not allow any transactions to be "backed up," the Delay/Re-Read Protocol must ensure that whenever any transaction performs a Write, that the transaction will be able to complete. For this reason, the initial Write of any transaction is critical.

We assume that when the transaction is submitted, its Read and Write set are known (actually the read set can be computed while processing the transactions, because the method does not use this information until all the Reads are done, but we need to know the Write set). This assumption has been made in SDD1 [BER], and is implicit in many locking protocols in order to determine whether to request a shared or exclusive lock. This does not place any restrictions on what transactions can read or write, but when the transactions is submitted, its read and write sets should be known, maybe by doing a prepass over the transaction before starting to process it. At the initial Write, the write set of the transaction is also recorded in the History File. If the write set of T^i is $\{x, y, z\}$ and the first write by T^i is on x , then it will be recorded in History File as $\hat{w}_i(x)\hat{w}_i(y)\hat{w}_i(z)W_i(x)$. All other Reads or Writes are simply recorded as $R_i(\text{entity-name})$ or $W_i(\text{entity-name})$.

A transaction which has performed at least one Write but has not yet terminated will be called *Active and Writing (AW)*, and the set of all active and writing transactions will be referred to as the *Active and Writing set (AWS)*.

Let us now present the Delay/Re-Read Protocol formally.

Let $x, y, z \in E$

The History File, H is maintained as a string over the alphabet²

² Actually, the "transaction number" $i \in [1, n]$ is not assigned by the CCP until the transaction's crit-

$$\Sigma = \{R_i(x), W_i(x), w_i(x) \mid i \in [1, n], x \in E\}$$

Let an ellipses (...) denote an arbitrary string over Σ (possibly of length zero).

Let $\text{AWS}(j)$ be the AWS just before T^j performs its critical write, $\text{TP}(j)$ the transaction process of T^j . The Delay/Re-Read Protocol is as follows:

```

Given a request for  $W_j(x(U))$ ,
1. for every  $i \in \text{AWS}(j)$  do
2.   { for every  $y \in U$  do
3.     { if  $H = \dots \hat{w}_i(y) \dots$ 
4.       then { if  $H \neq \dots W_i(y) \dots$ 
5.         then await  $W_i(y)$ 
6.         if  $H \neq \dots W_i(y) \dots R_j(y) \dots$ 
7.         then { instruct  $\text{TP}(j)$  to:
              a) re-read  $y$ 
              b) re-compute  $x(U)$ 
              c) re-request  $W_j(x(U))$ 
              halt
            }
        }
    }
  }
}
9. authorize  $W_j(x(U))$ 

```

Informally, the Delay/Re-Read Protocol ensures that there is no arc in G_S from T^j to T^i (where $(i, W_c, a) <_S (j, W_c, b)$). This is done by noticing the situation that

- a) T^j tries to use some $y \in E$ which
 - b) T^i expects to have written,
- and by ensuring that
- c) T^i has indeed performed that Write, and that
 - d) T^j 's Read (or Re-Read) of y occurs *after* T^i 's Write.

Condition b) is detected in line 3 of the Delay/Re-Read Protocol; the need for the preventative delay of step c) is detected in line 4 of the Delay/Re-Read Protocol, and; condition d) is either verified in line 6 of the Delay/Re-Read Protocol or is ensured via a corrective re-read by line 8 of the Delay/Re-Read Protocol.

Claim 5.1. The Delay/Re-Read Protocol is consistent.

Sketch of Proof. The above discussion illustrates that any $R_j(y)$ occurs *after* all $W_i(y)$ that may occur (for $i < j$). Thus, the hypotheses of Claim 3.3 are satisfied, and the resulting schedule is consistent.

ical write is authorized (so that $i < j$ indeed implies $(i, W_c, a) <_S (j, W_c, b)$). Consequently, the Reads must be initially recorded with some other transaction identification. Rather than exhibit this second identification and the obvious isomorphic mapping to the desired transaction numbers, we simply exhibit the transaction numbers throughout.

□

Claim 5.2. The Delay/Re-Read Protocol is deadlock-free.

Sketch of Proof. T^j is made to wait for T^i only if $(i, W_c, a) <_S (j, W_c, b)$. Since $<_S$ is a linear ordering, it follows that no deadlock can occur.

□

5. DISCUSSION

It may appear that the history string, H , grows without bound. However, there is a simple method by which we can prune H . We observe that once a transaction T^i has "terminated" (by having had its final Write authorized), then any prefix of H which involves only R_i , $\hat{w}_i$, and W_i , may be discarded. We further observe that the performance of a Re-Read, $R_j(x)$, obviates the need for any previous record of $R_j(x)$; thus H can be further pruned of such $R_j(x)$'s.

We may make a few observations concerning the "overhead" of the Delay/Re-Read Protocol. Overhead in the Delay/Re-Read Protocol is of two forms: "delay overhead" corresponding to the delay of a Write in step 5, and "re-read overhead" from step 7, which includes the re-computation of $x(U)$.

It is apparent that if H is pruned, as indicated above, then there is *no* overhead when there is only one active transaction. Moreover, there remains no overhead, even with multiple concurrent transactions, so long as they operate on disjoint sets of entities. Overhead increases only as the interaction among transactions increases, *vis-a-vis* increasingly overlapping Use Sets.

Some comparisons are possible between the Delay/Re-Read Protocol and Two-Phase Locking (although we make no attempt here at a full comparison). Delay/Re-Read Protocol often results in a greater degree of concurrency; consider the following example (where left-to-right vertical alignment indicates temporal ordering):

$$\begin{array}{lll} T_1 = R_1(x) & R_1(y) & W_1(x(x,y)) \\ T_2 = & R_2(x)R_2(z) & W_2(z(x,z)) \end{array}$$

Two-Phase Locking would force $R_2(x)$ to wait until after $W_1(x(x,y))$, effectively forcing serial execution, while the Delay/Re-Read Protocol permits full concurrency, as shown.

We have not yet indicated how the Delay/Re-Read Protocol may be applied in a distributed setting. In the simplest case, we envision that a $W_i(x)$ should make a local copy of the new value of x ; any subsequent $R_j(x)$, including Re-Reads of x , which are ordered by the CCP, will be made to read the recently-written T_i -site value of x . Thus, we have a database which distributes itself adaptively, according to the most recent Write.

This scheme also provides a solution to a practical implementation problem. Suppose T^i attempts $W_i(x(U))$ where the computation of $x(U)$ is expensive. It may

happen that T^i is forced to Re-Read various elements of U and, at each Re-Read, also re-compute $x(U)$. In the distributed model outlined above, T^i might view $W_i(x(U))$ as merely a request to write, without first performing the expensive computation. Only once the Write has been authorized by the CCP, does T^i proceed with the computation of $x(U)$, finally performing the Write locally. Meanwhile, all $R_j(x)$, which are directed by the CCP to Read the T^i -site value of x , are simply delayed by $TP(i)$ until T^i finally performs the Write of x .

6. CONCLUSION.

We have presented a new protocol for controlling concurrent access to a database which uses both preventative and corrective measures for maintaining consistency, and in so doing, permits a high degree of concurrency. The protocol is deadlock-free and accomplishes its "forward recovery" without the need for backup data, without the need for reversing the effects of any Writes, and without aborting transactions.

The present version of the Delay/Re-Read Protocol ensures *consistency vis-a-vis serialization*. We are working toward variations on the Delay/Re-Read Protocol which ensure even stronger forms of consistency.

REFERENCES

- [BAYa] Bayer, R., H. Heller and A. Reiser, "Parallesim and recovery in database systems", *ACM Transactions on Database Systems*, June 1980, pp. 139-156.
- [BAYb] Bayer, R. and M. Schkolnick, "Concurrency of operations on B-trees", *Acta Inf.*, vol 9-1, 1977, pp. 1-21.
- [BER] Bernstein, D. W., D. W. Shipman and J. B. Rothnie, "Concurrency control in a system for distributed databases (SDD-1)", *ACM Transactions on Database Systems*, March 1980, pp. 18-51.
- [ELL] Ellis, C. S., "Concurrency search and insertion in 2-3 trees", *Acta Inf.*, vol 14-1, 1980, pp. 63-86.
- [ESW] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger, "The notion of consistency and predicate locks in a database system", *Communications of the ACM* Nov 1976, pp. 624-633.
- [KED] Kadem, Z. and A. Silberschatz, "Controlling concurrency using locking protocols" in *Proceedings 20th IEEE Sym. on Foundations of Computer Science*, Oct 1979, pp. 274-285.
- [KUN] Kung, H. T. and J. T. Robertson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, June, 1981, pp. 213-226.
- [PAPa] Papadimitriou, C. H., "The serializability of concurrent database updates", *JACM*, Oct 1979, pp. 631-653.
- [PAPb] Papadimitriou, C. H. and P. C. Kanellakis, "On concurrency control by multiple versions" *Proceedings ACM SIGMOD Conference*, 1982, pp. 76-82.

- [SIL] Silberschatz, A. and Z. M. Kedem, "A family of locking protocols for database systems that are modelled as directed graphs", *IEEE Transactions on Software Engineering*, Nov 1982, pp. 558-862.
- [ULL] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, 1980.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUC-DCS-R-1145	2.	3. Recipient's Accession No.
4. Title and Subtitle The Delay/Re-Read Protocol for Concurrency Control in Databases		5. Report Date March 1983	
		6.	
7. Author(s) M. Dennis Mickunas and Pankaj Jalote		8. Performing Organization Rept. No.	
9. Performing Organization Name and Address Dept. of Computer Science University of Illinois Urbana, IL 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No.	
12. Sponsoring Organization Name and Address IBM Research Labs Highly Available Database Project San Jose, CA		13. Type of Report & Period Covered	
		14.	
15. Supplementary Notes			
16. Abstracts We present a new protocol, called the Delay/Re-Read Protocol, for controlling concurrent access to a database. The protocol uses a combination of preventative and corrective measures for maintaining consistency. The Delay-Re-Read Protocol is deadlock-free, requires no backup data, and does not abort either transaction or database writes.			
17. Key Words and Document Analysis. 17a. Descriptors protocol, concurrency, deadlock, database			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 13
		20. Security Class (This Page) UNCLASSIFIED	22. Price

UNIVERSITY OF ILLINOIS-URBANA



3 0112 103707102